A Meta-Modelling Approach to Feature Modelling

O. Rohlik[1], A. Pasetti[1, 2], K. Ekstein[2] , P. Chevalley[3]

[1] Institut für Automatik, ETH-Zentrum, Physikstr. 3,
CH-8092 Zürich, Switzerland
{rohlik, pasetti}@control.ee.ethz.ch
[2] P&P Software GmbH, C/O Institut für Automatik,
ETH-Z, CH-8092, Zürich, Switzerland
[3] European Space Agency, ESA-Estec, PO Box 299
2200 AG Noordwijk, The Netherlands
philippe.chevalley@esa.int

Abstract. Feature modelling is a key technology for product family engineering. In this paper, we describe a feature modelling technique that empowers product family engineers to define the framework within which they use feature modelling. Our technique offers three dimensions of flexibility. Firstly, it is based on a feature meta-meta-model and hence it allows users to define their own feature meta-model. Secondly, it allows users to exactly define the relationship between a family model and the models of the applications instantiated from the family. Thirdly, it allows users to define the visual rendering of a feature model. In the paper, we describe a tool that supports the proposed meta-modelling technique and a case study where the tool is configured to emulate an existing feature meta-model.

1 Background and Motivation

A *software product family* is a set of applications that can be built from a pool of shared software assets. Product families arguably offer the most successful path to software reuse [3, 7]. There is consequently a strong interest in technologies that further their use. Feature modelling is one of the key support technologies for product family engineering. It is in particular used to describe the target domain of a product family (the set of applications that can be instantiated from the family) and to define the range of potential configurations of the reusable assets of a product family.

In general, a *feature* is a characteristics of a system that is relevant to its end users. A *feature model* is a description of a set of features. Feature models are useful to identify all the features that can potentially appear in the applications to be instantiated from the family together with any constraints on their legal combinations (i.e. constraints on which combinations of features may appear in the same application). Feature models can be *instantiated*. A feature model is instantiated when a particular combination of its features is selected among all those which are defined by the feature model. If the feature model is used to describe the domain of a product family, its instantiation can be seen as an *application model*, namely as a description of a particular application within the family domain (an instance of the family).

Effective use of feature modelling techniques requires good tool support. A *feature modelling tool* should offer a GUI-based environment where users can create and maintain feature models. Additionally, if the tool is to serve as a *family modelling tool*, it should also cover the instantiation of the feature model. Manipulation of a feature model in a tool requires: (1) a definition of what exactly constitutes a feature model, and (2) a definition of how the feature model should be rendered visually. This can be restated by saying that manipulation of a feature model in a tool requires definition of a *feature meta-model* and a *feature display model*.

At present, only few feature modelling tools are available [8, 9, 10]. These tools are characterized by different feature meta-models and display models. This variety reflects a lack of consensus about the essential characteristics and visual appearance of feature models. The most common representation of feature models is through FODA-style feature diagrams [4]. Such diagrams have a tree-like structure where each node represents a feature and each feature may be described by a set of sub-features represented as children nodes. Various conventions have evolved to distinguish between mandatory features and optional features. Beyond this core set of characteristics for feature diagrams, there is however no agreement on the kind of information that should be attached to each feature, on the type of mechanisms that should be used to modularize large feature diagrams, and on the exact relationship between parent and children features. Existing feature modelling tools, in other words, are based on different – and usually incompatible – feature meta-models.

This lack of consensus on a feature meta-model is, in our view, intrinsic to the feature modelling problem (as opposed to being a temporary consequence of the novelty of the technique). Our experience from using feature modelling is that different users have different needs. At one extreme, there may be high-level domain designers who only need feature diagrams with simple tree structures where all features are of the same type with only one single value attached to them. At the other extreme, there may be software engineers who use feature models to capture the possible configurations of the reusable assets of a product family. These engineers may need to differentiate between different types of features with different semantics and with structured information attached to them; they may want to express complex constraints on feature combinations; and they may wish to break large feature models into independently manageable modules.

We believe that it is not appropriate to impose on such diverse users the same feature meta-model. In this paper, we present an alternative approach where the feature modelling technique is based upon a feature meta-meta-model and is parameterized with a user-defined feature meta-model. This makes it possible to build a feature modelling tool where users can define their feature meta-model and can use it to adapt the feature modelling tool to their particular needs.

In summary, our objective is to empower product family engineers to define the operational framework within which they use feature modelling. We do this primarily by allowing them to define their own feature meta-model but also offer them two further dimensions of flexibility. Firstly, we allow them to exactly define how a family model is instantiated by giving them control over the process whereby an application meta-model is created from a family model. Secondly, we allow them to define how a feature model is visually rendered.

The next two sections describe our feature modelling technique and our feature meta-meta-model. Section 4 describes a prototype tool which we developed to support our proposed feature modelling technique. Section 5 presents an example of how this tool can be configures to emulate a family modelling approach proposed by another author. Finally, section 6 considers related work.

2 Feature Modelling Approach

Our feature modelling approach was introduced to support product family modelling. It consequently covers three levels of modelling:

- The Product Family Meta-Modelling Level
- The Product Family Modelling Level
- The Application Modelling Level

At *family meta-modelling level*, the facilities available to describe a product family are defined. The family meta-model is fixed and family-independent. At *family modelling level*, a particular product family is described. A family model must be an instance of the family meta-model. Finally, at *application modelling level*, a particular application is described. The application model serves as a specification of the application to be instantiated from the family. The application model is an instance of the family model.

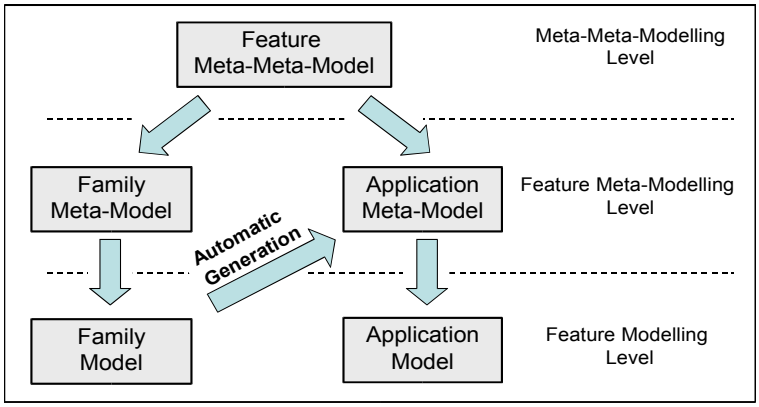

Fig. 1: Feature Modelling Architecture

Both the family model and the application model are represented as feature models. The former describes the mandatory and optional features that may appear in applications instantiated from the family (together with the constraints on their combinations). The latter describes the actual features that appear in a particular application. The application model is a feature model where all features are mandatory and where all variability has been removed.

Since both the family model and the application model are treated as feature models, it would in principle be possible to derive them both from a unique feature meta-model. However, the characteristics of these two models are rather different: the former aims to describe variability whereas the latter aims to describe a selection of features. We found that it is best to instantiate them from two distinct meta-models [1]. However, in

order to allow manipulation of both the family model and the application model within the same tool, their respective meta-models are derived from a common meta-meta-model. This is illustrated in figure 1. The family model (a feature model) is instantiated from the family meta-model (a feature meta-model). The application meta-model (a feature meta-model) is automatically generated from the family model. The application models are instantiated from the application meta-model. Both the application meta-model and the family meta-model are instances of a single meta-meta-model. With this concept, a product family is fully defined by its family model and by the program that generates the application meta-model from the family model.

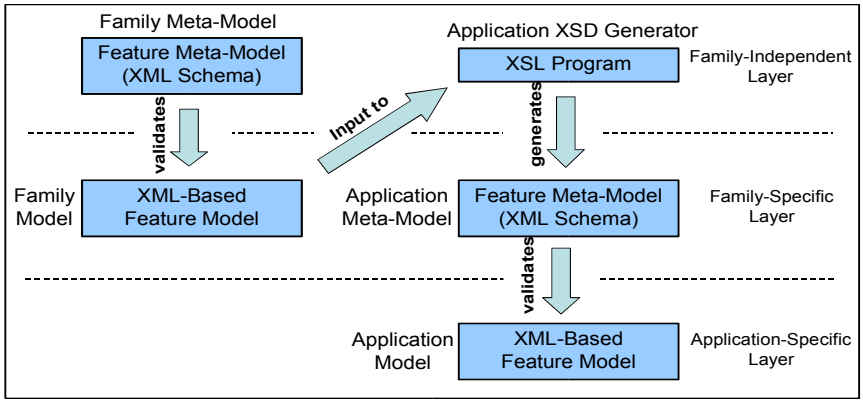

Fig. 2: XML-Based Feature Modelling Architecture

Feature models require the definition of a concrete syntax to express them. Several equivalent options are possible: UML-based, EMF-based, or XML-based. Our work uses an XML-based approach. This choice was dictated by pragmatic considerations: the availability of reusable components for the construction of a tool, the stability of XML standards, and the familiarity of prospective users with XML-based processing. Additionally, as shown in [1], an XML-based approach allows users to quickly prototype their family models using standard XML editors.

In our approach, feature models are expressed using XML-based languages and the relationship of instantiation between a model and its meta-model is expressed by saying that the XML-based model must be validated by the XML Schema that represents its meta-model. The modelling architecture of figure 1 can then be recast as in figure 2. Both the family and the application models are expressed as XML-based feature models but they are instantiated from two different meta-models that take the form of XML Schemas. The application meta-model is automatically generated from the family model by an XSL program (the *Application XSD Generator*). Note that, although not shown in figure 2, both the family meta-model and the application meta-model are constrained by a meta-meta-model (the top-level box in figure 1).

With the architecture of figure 2, a family is completely defined by its feature model (an XML document) and by the Application XSD Generator (an XSL program).

The key factors of our approach are therefore the representation of a feature model as an XML document, and the representation of its meta-model as an XML Schema that

validates the XML document. XSD (the language used to express XML Schemas) is not very expressive but we found that it is sufficiently powerful to express a wide array of feature model structures. Indeed, as discussed in section 5, we were able to map three different feature meta-models (including the very recent one from [9]) to our approach without any loss of expressiveness.

3. The Feature Meta-Meta-Model

In our concept, a feature meta-model is expressed as an XSD document (an XML Schema). However, not all XSD documents represent valid feature meta-models. In order to formalize the notion of "valid feature meta-model", the notion of feature meta-meta-model is introduced. Valid feature meta-models must be instances of a feature meta-meta-model (the top box in figure 1).

Conceptually, the feature meta-meta-model enforces a certain structure for all possible feature models. This structure can itself be represented as a feature diagram that is shown in figure 3 using the notation introduced in [5]. As shown in the figure, a feature model has a tree-like structure. At the top level, a *feature model* has a *name* and one *node*. Each node can, recursively, have other nodes as its children. Nodes are characterized by a mandatory *name* that uniquely identifies the type of the node and by minimal and maximal *cardinalities*. Nodes may have zero or more *property sets* attached to them that define sets of logically related *properties.* A property in turn defines a typed value that can be associated to the node.

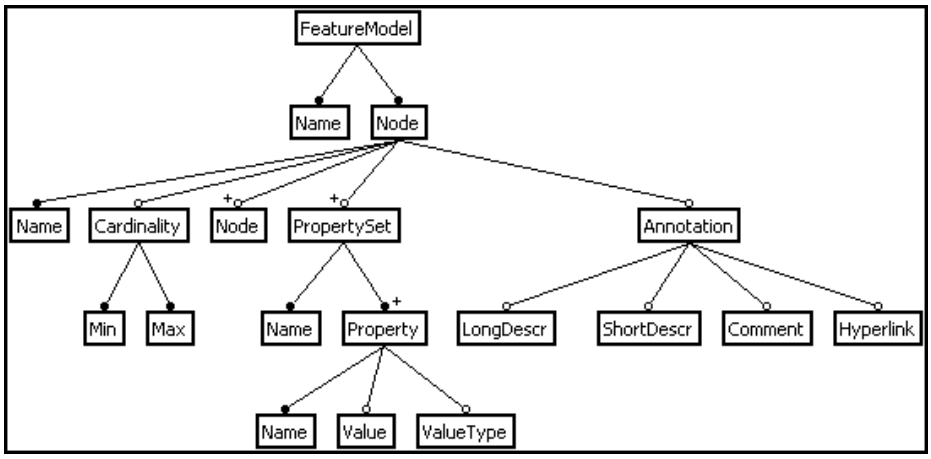

Fig. 3: Conceptual Structure of a Feature Model

Generally speaking, nodes in a feature meta-model are used to represent classes of features with the same semantics. Thus, for instance, a meta-model for the feature modelling approach introduced by [9] would use distinct nodes (i.e. nodes with distinct names) to represent the root feature, solitary features, solitary references, etc.

The property mechanism is introduced to allow typed values to be associated to a node in a feature model. The simplest option for doing so would have been to define a fixed structure whereby to each node may be associated one – or perhaps a small

number of – properties. This approach however is too rigid, both because different meta-models have different needs and because, within the same meta-model, it is usually desirable to associate different sets of properties to different nodes. The next simplest approach is to allow the feature meta-model to specify which sets of properties should be associated to which type of node. This approach is still too rigid as it does not allow different sets of properties to be associated to the same kind of nodes. We have therefore adopted a more general approach where all properties required within a model are gathered together in property sets. A property set allows a set of logically related properties to be manipulated as a single entity. The meta-model specifies which property sets may be associated to which types of node.

The feature meta-meta-model recognizes one particular kind of property set (the *annotation* property set). This property set is intended to gather together properties that should be directly manipulated by a feature editing tool. In practice, it contains properties that are used to visually decorate the feature model representation.

Our feature meta-meta-model enforces the structure of figure 3. It additionally offers a mechanism to express constraints on how nodes and properties may be combined. The technical details of how this is done are rather complex (they are described in a technical report available from [2]) but the general idea is outlined below.

Our feature meta-meta-model assumes that a feature meta-model is encoded as an XML Schema and that each XML Schema element describes a *feature model element*. With reference to the discussion above and to figure 3, the possible feature model elements are *node*, *cardinality*, *property set*, and *property*. These feature model elements are the elementary "building blocks" that the meta-meta-model makes available for describing a feature meta-model and, through it, a feature model.

By default, XML Schemas are validated against the `XMLSchema.xsd` schema which is defined by the W3 consortium and which defines the XSD language. We have treated the problem of creating a feature meta-meta-model as the problem of creating a schema that incorporates the standard `XMLSchema.xsd` schema but adds to it the additional constraints specific to our problem. The first such constraint is that XML Schema elements must have a mandatory attribute representing the feature model element which that schema element describes. This establishes the link between the XML Schema elements and the feature model elements. The second constraint enforces the relationships between feature model elements illustrated in figure 3. Thus, for instance, it constrains elements that represent properties to be contained within elements that represent property sets. Or it constrains elements that represent nodes to be contained within other elements that represent nodes. And so forth.

In practice, the new XML validating schema representing our feature meta-meta-model is obtained by modifying some of the rules of the `XMLSchema.xsd` schema and by embedding within the `XMLSchema.xsd` schema some additional rules based on the Schematron schema language. The latter was needed because the XML Schema, by itself, is not powerful enough to express all the structural and syntactical constraints required for our purpose. The result is a feature meta-meta-model that is implemented as a single document representing a valid XML Schema. As such, it can also be used in a standard XML editing tools to provide code completion facilities that can help users to construct their feature meta-models.

4. The XFeature Meta-Modelling Tool

We have developed XFeature as a tool to implement the feature modelling approach presented above. Its purpose is to support family modelling activities by providing a GUI-based environment where users can define both the model of a family and the models of applications within the family.

At its most basic, XFeature does not differentiate between family and application models. It simply offers an environment where users can construct a feature model as an instance of a certain meta-model. XFeature is a meta-modelling tool because it allows users to define the feature meta-model they wish to use in a certain editing session (the feature meta-model has to comply with the feature meta-meta-model defined in section 3). XFeature can thus be used to perform both family modelling (by parameterizing it with a feature meta-model that represents a family meta-model) and application modelling (by parameterizing it with a feature meta-model that represents an application meta-model).

We expect different users to want to use different graphical notation to represent their feature models. In order to provide flexibility in this respect, the XFeature tool is also designed to be parameterized with a *display model*. The display model defines how the individual elements of the feature model (the nodes of the feature diagram, the lines linking parent to children nodes, etc) should be rendered graphically. The resulting architecture of the XFeature tool is shown in figure 5. The figure indicates that the feature meta-model is expressed as an XML Schema and the display model is expressed as an XML document. Note that the display model is constrained to be an instance of a display meta-model expressed, as usual, as an XML Schema.

Fig. 5: XFeature Tool Structure

Figure 5 shows the essential structure of XFeature which is that of a pure feature modelling tool completely decoupled from its expected usage as an aid in modelling product families. In reality, for obvious reasons of convenience, although the editing facilities of the tool are indeed independent of the distinction between family and application models, XFeature also provides facilities to make the manipulation of feature models that represent family and application models easy. Its full structure is shown in figure 6. The tool is parameterized with the following elements:

- A *family meta-model* (an XML Schema representing a feature meta-model)
- An *application XSD generator* (an XSL program that translates a feature model representing a family model into an XML Schema representing the feature meta-model for the application model)
- A *display model* (an XML document that defines how each element in a feature model should be displayed)
- An *application display model generator* (an XSL program that translates the

display model of a family into the display model for the applications instantiated from the family)

The above elements are collectively called the *XFeature Configuration Files*. Taken together, they define a particular feature modelling approach.

For completeness the figure also shows two additional plug-ins for the tool that, at the time of writing, are still under development. They relate to the expression and enforcement of global composition rules on the feature models. Global composition rules are constraints on the combinations of non-adjacent features in a feature model. The approach to handling such constraints is as defined in [1]. The two plug-ins represent a global constraint meta-model (an XML Schema that defines an XML-based language to define global constraints) and a global constraint model compiler (an XSL program that processes a global constraint model and generates a program that verifies whether a certain feature model satisfies the global constraints).

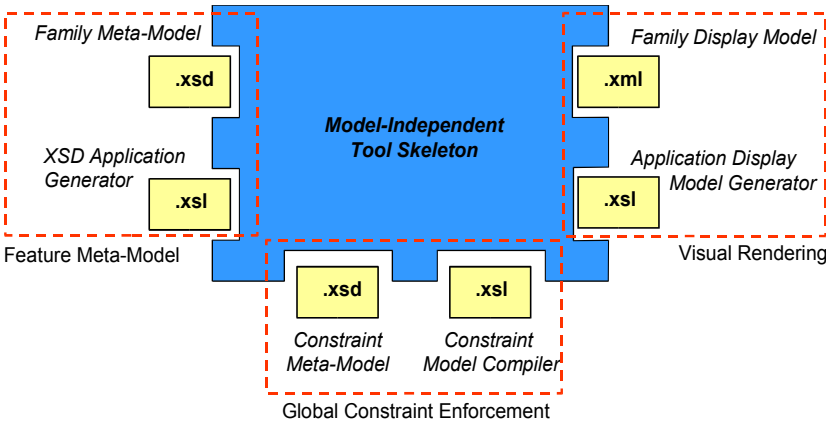

Fig. 6: High-Level Architecture of XFeature Tool

The expected mode of use of XFeature is as follows. First, users with common needs define their approach to family modelling. This approach is then formally expressed as a feature meta-model implemented as an XML Schema that must be validated by the XFeature meta-meta-model. In the next step, the family instantiation process is considered. This entails the definition of an application modelling approach. This is formalized through the definition of an XSD Application Generator program. Finally, the visual rendering of the family and application models are defined leading to the definition of the Family Display Model and the Application Display Model Generator. The resulting configuration files are loaded into the XFeature environment and transform it into a tool than enforces the selected family modeling approach.

Figure 7 shows an example of a family model constructed with XFeature. The model shown in the figure has the same semantics as the family model of reference [9]. The figure shows the feature model as a tree (broken up, for ease of manipulation, into a root tree and three substrees). The visual rendering of the feature model elements is entirely determined by the display model. *Nodes* can be rendered either as rectangular box or as circular dots. *Node cardinalities* are displayed in dark boxes at the top right-

hand corner of the nodes. The nodes can be connected by straight lines, they can be attached to each other, or they can even be superimposed on each other. Different colours may be used to distinguish between different kinds of nodes. The position of the nodes can be manually adjusted by the user. The *properties* and their *property sets* are displayed and edited in the property sheet at the bottom of the screen. The window at the left shows an outline view of the feature model. It is possible to switch to the XML view of a model where users can directly edit the raw XML representation of their feature model.

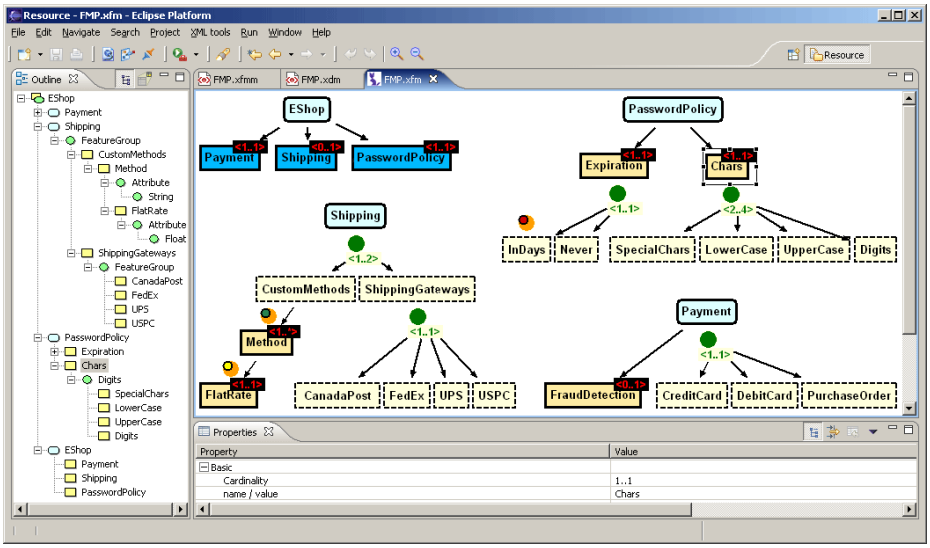

Fig. 7: XFeature Screenshot

The feature diagram in figure 7 may look unfamiliar because, although we can easily emulate the classical FODA feature meta-model, we cannot at present emulate the classical FODA representation of feature groups (we cannot yet draw arcs joining adjacent features). In figure 7, we use green dots to represent feature groups to which we attach a maximal and minimal cardinality representing the maximum and minimum number of features from within the group that can be selected. Thus, for instance, features “CreditCard”, “DebitCard” and “PurchaseOrder” belong to a group of features of which exactly one must be selected.

The XFeature tool is built as an Eclipse plug-in. Model parsing and editing is done using standard XML components. The graphical rendering is implemented through Eclipse's *Graphical Editing Framework* (GEF). The tool is available as free and open software from the XFeature web site [2].

5. XFeature Configuration Example

We claim that our approach subsumes that of other authors because it shifts the focus of modelling from the feature meta-model level to the feature meta-meta-model level. One way to verify this claim is to show that the XFeature tool can be configured to emulate other feature modelling approaches. As an example, we describe how

XFeature was configured to emulate the modelling approach of reference [9]. In general, the configuration of XFeature to emulate a particular family modelling approach is done in 5 steps:

1. The feature modelling elements used to describe family models are identified and are mapped to *nodes*, *property sets*, and *properties*. This leads to the identification of the basic elements of the *family meta-model XML Schema*.
2. The constraints on the combinations of these modelling elements are identified and are encoded using the XSD language in the *family meta-model XML Schema*.
3. The visual rendering desired for each feature modelling element is defined and encoded in the *display model*.
4. Steps (1) and (2) are repeated for the target application meta-model leading to the definition of the *application XSD generator*.
5. Step (3) is repeated for the target application meta-model leading to the definition of the *application display model generator*.

In the case of reference [9], the feature modelling elements are: RootFeature, SolitaryFeature, SolitaryReference, FeatureGroup, GroupedFeature, and GroupedReference. We have mapped them all to nodes and we have used the display model to specify renderings that visually differentiate them, For instance, FeatureGroups are represented by dashed boxes, GroupedFeatures by green dots attached below their parent feature, SolitaryFeatures by solid boxes, etc. Figure 7 shows an example of a family modelled in XFeature using this configuration.

In reference [9], each feature can have an attribute and the attribute has a type, a value, and a default value. Values can be integer, string or float. We have mapped both attributes and their possible types to nodes. These nodes simply model the existence of an attribute and its type. Thus, for instance, a solitary feature with a string attribute is modeled as a node of type “solitary feature” which has a sub-node of type “attribute” which in turn has a sub-node of type “string”. In order to store the value and default values associated to the attributes, three property sets have been defined (one each for string, integer and float values). Each property set has two properties representing the value and the default value. The attribute nodes are represented as small dots positioned above a feature with a colour that defines their type (red for integer attributes, green for string attributes, and yellow for float attributes).

In reference [9], a specialization mechanism is introduced. We did not model it explicitly but could have done so by following the same approach we adopted for the macro extension mechanism of reference [1]. This would have required the definition of two additional types of node representing, respectively, a feature that is specialized and the features that are removed during the specialization process.

In reference [9], several constraints are defined on the combination of the family-level feature modelling elements. For instance, children of a FeatureGroup can only be GroupedFeatures or GroupedReference; or GroupedFeatures have a fixed cardinality. Such constraints were encoded in the XSD language.

The mapping of a family model to the application meta-model was done as follows. SolitaryFeatures were mapped to features that can be instantiated as many times as allowed by the cardinality of the original SolitaryFeature. FeatureGroups represent a set of legal combinations of GroupedFeatures. Each such combination is identified

during the mapping process and separately represented in the application meta-model. Reference features (GroupedReference and SolitaryReference) behave like pointers to feature subtrees. All such references are resolved during the transformation process from family model to application meta-model. Attributes are mapped to properties. Finally, the application display model generator was constructed to maintain the visual conventions defined at family level. Figure 8 shows an example of an application instantiated from the family shown in figure 7. As usual, the tool offers both a graphical view (main window) and an outline view (left-hand window).

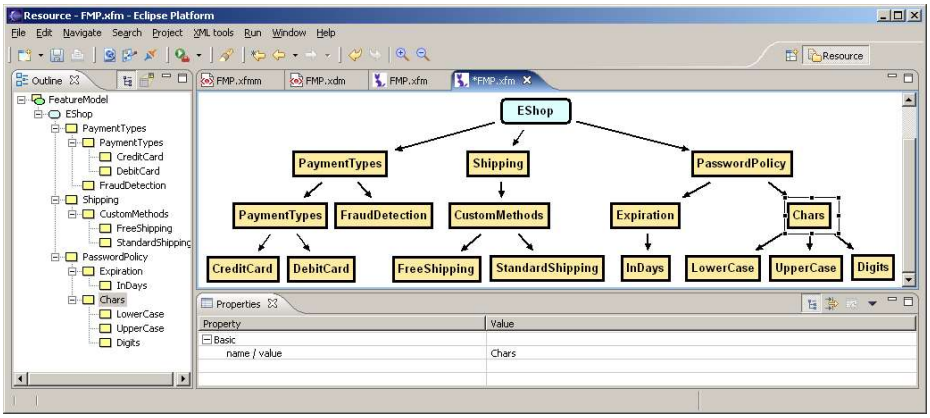

Fig. 8: Example of application instantiated from family model of fig. 7

The configuration outlined above is encapsulated in four files. Once these files are loaded into the XFeature tool, they turn it into an emulator of the modelling tool of reference [9]. Replacing the configuration files completely changes the behaviour of the tool. In fact, in addition to the configuration emulating the family modelling approach of reference [9], we also defined two further XFeature configurations implementing two different modelling approaches. The predefined tool configurations are provided with the standard XFeature delivery [2]. They demonstrate its flexibility in emulating different modelling approaches and they serve as blueprints for users to create their own configuration files adapted to their own special needs.

6. Related Work

The origins of the work presented in this paper are found in [1] where we proposed an XML-based approach to feature modelling. The present paper extends this work by adding a meta-modelling dimension. The modelling approach originally proposed in [1] has become one instance of the more general meta-modelling approach proposed in the present paper.

Several authors have proposed feature modelling approaches [4, 5, 6, 8, 9] but their models stop at the feature meta-model level. To our knowledge, we are the only ones to have explicitly aimed at defining a common feature meta-meta-model with the goal of encompassing the feature meta-models proposed by others.

Our approach is also distinctive in being XML-based. Other authors have generally

preferred to work within the Eclipse Modelling Framework (EMF). The EMF is especially effective at modelling and editing but has weaker support for transformations, whereas the opposite is the case for the XML framework. Ultimately, however, the two frameworks are equivalent in their expressiveness. We chose to stay in the XML world in order to take advantage of the large amount of high-quality XML processing components and tools. We also hope that the popularity of XML technology will encourage adoption of our approach on the part of practitioners.

The fact that users have the option to define their own feature meta-model is the main advantage of our approach. Since we implement meta-models as XML Schemas, a technology with which many users are likely to be familiar, it will be easy for them to benefit from this option and fully exploit the flexibility of our approach. This is a further advantage of our choice of working within the XML framework.

Finally, we have emphasized the graphical rendering of feature diagrams. This is in contrast to other authors who privilege the outline view. Conceptually, this is not a significant difference. However, we believe that practical manipulation of large feature diagrams requires something more than just a tree outline. Indeed, the improvement of the user interface and the provision of better facilities to manipulate feature diagrams (save/restore facilities for subtrees, collapsing of subtrees upon mouse click, etc) are one of our current lines of work on the XFeature tool.

In summary, we believe that our approach raises the level of abstraction at which feature modelling is done and that, in so doing, it provides a common framework within which other feature modelling approaches can be formally expressed and where they can be easily extended and compared.

References

1. Cechticky, V., Pasetti, A., Rohlik, O., Schaufelberger, W.: XML-Based Feature Modelling, in Bosch, J., Krueger, C. (eds), Software Reuse: Methods, Techniques, and Tools (ICSR), LNCS Vol. 3107, Springer-Verlag, 2004
2. XFeature Web Site, http://www.pnp-software.com/XFeature/
3. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, Addison-Wesley, 2003
4. Kang, K., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: FORM: A Feature-Oriented Reuse Method. Technical Report, Pohang University of Science and Technology, 1998
5. Czarnecki, K., Eisenecker, U.: Generative Programming – Methods, Tools, and Applications. Addison-Wesley, 2000
6. Beuche D.: Composition and Construction of Embedded Software Families, Doctoral Dissertation, Univ. of Magdeburg, Dec. 2003
7. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns, Addison-Wesley, 2001
8. Captain Feature Web Site, https://sourceforge.net/projects/captainfeature/
9. Antkiewicz, M., Czarnecki, K.,: FeaturePlugin: Feature Modeling Plug-In for Eclipse, OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop, Oct. 24-28, 2004, Vancouver, British Columbia, Canada
10. Pure::Variants Website, http://www.pure-systems.com